

Correction Trees as an Alternative to Turbo Codes and Low Density Parity Check Codes

Jarosław Duda, Paweł Korus, *Student Member, IEEE*,

Abstract—The rapidly improving performance of modern hardware renders convolutional codes obsolete, and allows for the practical implementation of more sophisticated correction codes such as low density parity check (LDPC) and turbo codes (TC). Both are decoded by iterative algorithms, which require a disproportional computational effort for low channel noise. They are also unable to correct higher noise levels, still below the Shannon theoretical limit. In this paper, we discuss an enhanced version of a convolutional-like decoding paradigm which adopts very large spaces of possible system states, of the order of 2^{64} . Under such conditions, the traditional convolution operation is rendered useless and needs to be replaced by a carefully designed state transition procedure. The size of the system state space completely changes the correction philosophy, as state collisions are virtually impossible and the decoding procedure becomes a correction tree. The proposed decoding algorithm is practically cost-free for low channel noise. As the channel noise approaches the Shannon limit, it is still possible to perform correction, although its cost increases to infinity. In many applications, the implemented decoder can essentially outperform both LDPC and TC. This paper describes the proposed correction paradigm and theoretically analyzes the asymptotic correction performance. The considered encoder and decoder were verified experimentally for the binary symmetric channel. The correction process remains practically cost-free for channel error rates below 0.05 and 0.13 for the 1/2 and 1/4 rate codes, respectively. For the considered resource limit, the output bit error rates reach the order of 10^{-3} for channel error rates 0.08 and 0.18. The proposed correction paradigm can be easily extended to other communication channels; the appropriate generalizations are also discussed in this study.

Index Terms—error correction coding, convolutional codes, sequential decoding

I. INTRODUCTION

Approaching the Shannon limit in forward error correction inherently involves operating on data blocks of rapidly increasing size. Alongside this growth, the number of valid codewords increases exponentially. As a result, the correction process, which involves finding the closest valid codeword to the received one, quickly becomes impractical. Error correction methods are considered practical if they are able to find an approximation of the optimal correction in reasonable time. In general, there are two main approaches to the problem [10]. The first is to spread pseudo-random verification bits uniformly over the transmitted message. This approach is

adopted by convolutional codes [6] and their derivatives, such as turbo codes (TC) [3]. The second approach uses linear block codes with very low degrees of vertices. Such codes are known as low density parity check (LDPC) or Gallager codes [8].

Both LDPC and TC have their limitations. The former are defined by very large sparse matrices, whose generation and control during the correction process requires significant computational resources. Both LDPC and TC are decoded by iterative algorithms, usually with a pre-defined number of iterations. As such, they are characterized by a nearly constant decoding cost, regardless of the noise level actually observed in the channel. Due to the increasing capabilities of modern hardware decoders, this cost is now becoming acceptable, and the methods emerge as a replacement for conventional convolutional codes.

The family of convolutional codes is in practice limited to relatively small spaces of possible system states, which stems from the inconveniences caused by the intrinsic convolution operation. The typical size of the system state space is at most 2^{16} . As a result, wrong correction patterns frequently generate correct system states, and instead of a complete correction, the decoder is merely able to reduce the number of errors in the message. The TC addressed this issue by repeating the encoding process on an interleaved version of the data blocks. They use half of the redundancy in every step, and require to consider the entire system state history. Hence, the correction of even low error rates already requires relatively long times. For convolutional codes, there is a computational cut-off rate [9] above which the expected correction time per symbol grows to infinity for infinite data streams. We use finite fixed length data frames (mainly 1kB), which enables efficient operation with rates up to the Shannon limit.

In this study, we analyze the correction potential of adopting large spaces of possible system states in a convolutional-like coding process. We will show that such an approach leads to very efficient correction algorithms, which enable correction arbitrarily close to the Shannon limit. We consider the space of at least 2^{64} system states, where the probability of system state collision for a wrong correction pattern becomes negligible ($2^{-64} \approx 5 \cdot 10^{-20}$). This approach becomes tractable by designing a decoder which processes a very small fraction of the state space. We will show that the decoding process can be made practically cost-free for small error rates, and that no additional correction limits arise. Theoretically, a once encoded message can always be nearly completely corrected under the Shannon limit; however, in this limit the decoding cost grows to infinity.

Due to the negligible system state collision probability, the correction structure practically no longer contains cycles, and

J. Duda is with the Institute of Physics, Jagiellonian University, ul. Reymonta 4, 30-059 Kraków, Poland. E-mail: dudaj@interia.pl

P. Korus is with the Department of Telecommunications, AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków, Poland. E-mail: pkorus@agh.edu.pl

The research leading to these results has received funding from the INDECT project funded by European Community's Seventh Framework Programme (FP7 / 2007-2013) under grant agreement no.[218086].

becomes a tree instead. The nodes of this tree represent a correction path up to a given position in the data stream. This approach imposes sequential decoding, which considers the most probable branch of the tree in each decoding step. The probability of a tree branch is reflected in the *weights* of the corresponding tree nodes. The greater the weight, the more probable the corresponding node. For the purpose of efficient access to successive prospective correction paths, we use a heap to store the nodes of the tree. The heap operations have logarithmic complexity, which constitutes a significant improvement over commonly used linear-complexity structures, such as the stack.

For large error concentrations, the proposed correction algorithm may require impractically large numbers of necessary decoding steps. For any acceptable total number of decoding steps, there is a nonzero probability that the given step limit is insufficient. We refer to such situations as *critical error concentrations* (CEC). In this study, we provide a theoretical methodology for quantitative analysis of the error concentrations, their probability distributions, and the expected number of decoding steps in which the decoder can deal with them. We will focus on the binary symmetric channel (BSC) and discuss different channel noises.

Susceptibility to critical error concentrations stems from the sequential nature of data stream processing. In traditional convolutional codes, where the state space is strongly limited, this problem can be mitigated by guessing the correct state after a concentration of errors. To some extent, error bursts in the channel can be dispersed with the adoption of stream interleavers. In order to address this issue, we adopt two techniques. Firstly, we construct a systematic code in order to bound the output error rate by the input error rate. Secondly, we design the scheme so that it would be possible to process the stream from both the beginning and the end. After reaching the same place in the stream, the correction paths are merged. With bidirectional correction, a data block will remain essentially damaged if there are at least two independent CECs. The probability of such an event is approximately equal to the square of the probability of a single CEC.

In summary, the main contribution of our work addresses the following issues. Firstly, we consider a much larger system state space and design of a tailored fast coding and decoding scheme. Secondly, we adopt a logarithmic-time heap structure instead of a commonly used linear-time stack. Thirdly, we propose an efficient state transition procedure, optimized for both forward and backward processing. We also design a dedicated mechanism, based on look-up tables, which enables rapid pruning of invalid tree branches. The decoder considers just the correction patterns which are guaranteed to lead to the allowed system states.

The paper is organized as follows. Section II describes the proposed encoding and decoding algorithms. Theoretical analysis of the decoding performance and the impact of error concentrations is presented in Section III. The results of experimental evaluation are shown in Section IV. Further perspectives for adapting to different communication channels are briefly presented in Section V. A discussion of the applicability of the proposed scheme and final conclusions are

presented in Section VI.

II. THE CORRECTION-TREE ALGORITHM

This section describes the principles of the proposed correction algorithm. We begin with an explanation of the encoding procedure and a corresponding forward correction algorithm. We then extend the algorithm to a bidirectional case. A theoretical foundation for the utilized error correction approach can be found in Section 9 of [4].

The following notation is used in the paper:

| | |
|-------------|--|
| \propto | proportional to |
| \oplus | bit-wise exclusive disjunction |
| $\&$ | bit-wise logical conjunction |
| \tilde{p} | $= 1 - p$ |
| \lg | \log_2 |
| $h(p)$ | $= -p \lg(p) - \tilde{p} \lg(\tilde{p})$ |
| $\#A$ | cardinality of set A |
| n | total number of bits in encoded symbols |
| k | number of payload bits in encoded symbols |
| R | $= n - k$ - the number of redundancy bits |
| k/n | code rate (excluding the final state) |
| p_d | $= 1 - 2^{-R}$ - probability that redundancy bits would accidentally agree |
| l | number of symbols in data frame |
| ϵ | probability of bit flip in a binary symmetric channel (BSC) |
| p_d^0 | $= 1 - 2^{-nh(\epsilon)}$ - BSC Shannon limit for p_d |
| cn | current node of the correction tree |
| $cn.d$ | data symbol position associated with cn |
| $cn.p$ | parent node of cn |
| $cn.s$ | system state associated with cn |
| $cn.e$ | correction pattern associated with cn |

A. Encoding Procedure

The operation of the encoder begins with initializing the system state and the state transition tables. The algorithm operates sequentially on successive symbols from the data stream. Upon reception of a new symbol, the encoder performs symbol-dependent transition of the system state and extracts the redundancy as a certain number of the least significant bits of a binary representation of the posterior system state. This redundancy is transmitted together with the symbol in a systematic manner, i.e., by concatenation of the payload and the redundancy information.

There are two important consequences of this approach. Firstly, the transmitted stream is equipped with uniformly distributed redundancy information. Secondly, the redundancy of all transmitted symbols is intuitively connected by the system state. As a result, it becomes a resource which is shared among the symbols. In general, the amount of the introduced redundancy and thus the code rate can be chosen arbitrarily. In this study, we focus on constant redundancy per symbol and use code symbols represented by $n = 8$ bits.

After processing all the symbols, the final encoder state should be communicated to the decoder. This state is required by the backward correction algorithm. When only forward

Algorithm 1 Encoding procedure

Require: $k, n, R := n - k$
Require: $\mathbf{x} := x_1, x_2, \dots, x_l$ // Data stream n -bit symbols with R bits zeroed
 $s_0 \leftarrow$ Initialize system state
 $r_m \leftarrow 2^R - 1$ // Redundancy extraction mask
for $i = 1 \rightarrow M$ **do**
 $s_i \leftarrow \text{updateState}(s_{i-1}, x_i)$
 $y_i \leftarrow x_i + (s_i \& r_m)$
end for
 Transmit $\mathbf{y} := y_1, y_2, \dots, y_M$ // Encoded stream of n -bit symbols

correction is used, this step can be omitted at the cost of potential corruption of the last portion of the symbols.

The operation of the encoder is summarized in Algorithm 1. The details of the state transition procedure will be presented in Section II-D.

B. Forward Correction Procedure

Principally, the decoder repeats the symbol processing procedure performed by the encoder. Upon reception of a new symbol, it performs a symbol-dependent state transition. From the posterior system state, it compares the resulting redundancy bits with the received ones. As a result, for error-free transmission, the decoding process is practically cost-free and takes the same amount of time as the encoding.

Assuming R redundancy bits per symbol have been used, the probability of detecting an error is $p_d = 1 - 2^{-R}$ per node. When an error is successfully detected, the corresponding node of the correction tree is not analyzed any further. The decoder then considers other correction paths, starting from the most probable ones. We will later show that for a given p_d and channel error rate below the Shannon limit, the decoder will be able to prune correction tree branches faster than they are created.

We refer to the posterior state which passed the redundancy check as *allowed*, and the failing one as *forbidden*. When a forbidden state is reached, the decoder needs to go back and apply a new correction to the current or one of previous symbols. Intuitively, if the current symbol is being considered for the first time in the current correction path, the most likely error pattern is a single bit error in the current symbol. In general, the decoder will select the next correction path for consideration by choosing a correction tree node with the greatest weight. The weights are calculated cumulatively from the beginning of the correction process and decremented each time a next bit is considered as corrupted.

A proper sequence of traversing the correction tree branches is ensured by using a heap for storing the tree nodes. The heap is ordered by the weights of the nodes. For the sake of the decoding performance, the decoding algorithm considers only feasible corrections, i.e., the ones that lead to the allowed states. This can be achieved by using a properly designed state transition procedure and dedicated state transition tables. Hence, it is possible to significantly reduce the use of the heap, which is the most time-consuming operation in the process. The forward error correction procedure is presented in a simplified way in Algorithm 2. The complete algorithm is included as a source code along with the paper [1].

Algorithm 2 Operation of the decoding process with forward correction (simplified)

Require: $k, n, R := n - k$
Require: s_f // Final system state
Require: $\mathbf{y} = y_1, y_2, \dots, y_l$ // Data stream n -bit symbols
 Initialize correction lookup table
 $s \leftarrow$ Initial system state
 $cn \leftarrow \text{null}$ // The current node
 $pn \leftarrow$ before the first symbol // Parent node
repeat
 repeat
 // Consider $pn.e$ -th correction of parent node
 Set cn as $pn.e$ -th child of pn
 $cn.e \leftarrow 0$ // Set it to null correction
 $cn.s \leftarrow s$ // Remember current system state
 Remember cn on a list of visited nodes
 $s, c \leftarrow \text{updateState}(s, y_{cn.d})$ // System state transition
 $pn \leftarrow cn$ // Set current node as parent for the next cycle
 // Finish if forbidden state is detected or there are no more symbols
until $c = \text{false}$ **or** $cn.d > \text{last symbol}$
 // Set to next correction pattern passing verification
 $cn.e \leftarrow$ from pre-initialized lookup table
 pushHeap(cn) // Push the item on the heap
 $pn \leftarrow \text{popHeap}()$ // Get new node to consider
if pn is null correction child **then**
 // Create first sibling of pn and add it to heap
 $cn \leftarrow pn.p$ // Set cn as parent of pn
 $cn.e \leftarrow$ from pre-initialized lookup table
 pushHeap(cn) // Push the item on the heap
end if
 Increase the number of error for considered node...
 ...in the list of visited nodes and push it to heap
 $s \leftarrow \text{updateState}(pn.s, y_{pn.d} \oplus pn.e)$ // System state transition
 // Finish if after processing the last symbol the system state is correct
until $cn.d > \text{last symbol}$ **and** $s = s_f$

The considered correction method is illustrated schematically in Fig. 1. The numbers of the nodes represent the sequence in which the decoder traverses the correction tree. The labels of the state transitions denote the symbol and the currently considered correction of that symbol, i.e., c_0 denotes a null correction vector, c_1 the most probable correction vector, c_2 the second probable correction vector, etc. In the presented example the s_3 , s_6 and s_{10} symbols are corrupted. For s_3 , the decoder immediately notices a forbidden system state and the most probable correction turns out to be the proper one. For s_6 , the decoder detects a forbidden state at first, but then fails to correct it. After considering a number of wrong corrections, the most probable turns out to be the second correction of s_6 . For the last symbol, s_{10} , the decoder does not rely on its redundancy information and uses the knowledge of the final system state to detect wrong correction vectors with probability close to 1.

C. Weighting the Correction Tree Nodes

In this study, we consider the transmission channel to be a binary symmetric channel, i.e., for each of the transmitted bits, there is a constant probability ($\epsilon > 0$) that the bit will be received as corrupted. The number of possible errors in a N bit sequence can be asymptotically estimated as:

$$\binom{N}{\epsilon N} \approx 2^{Nh(\epsilon)} \quad (1)$$

If we were to build a tree of all typical corrections for j

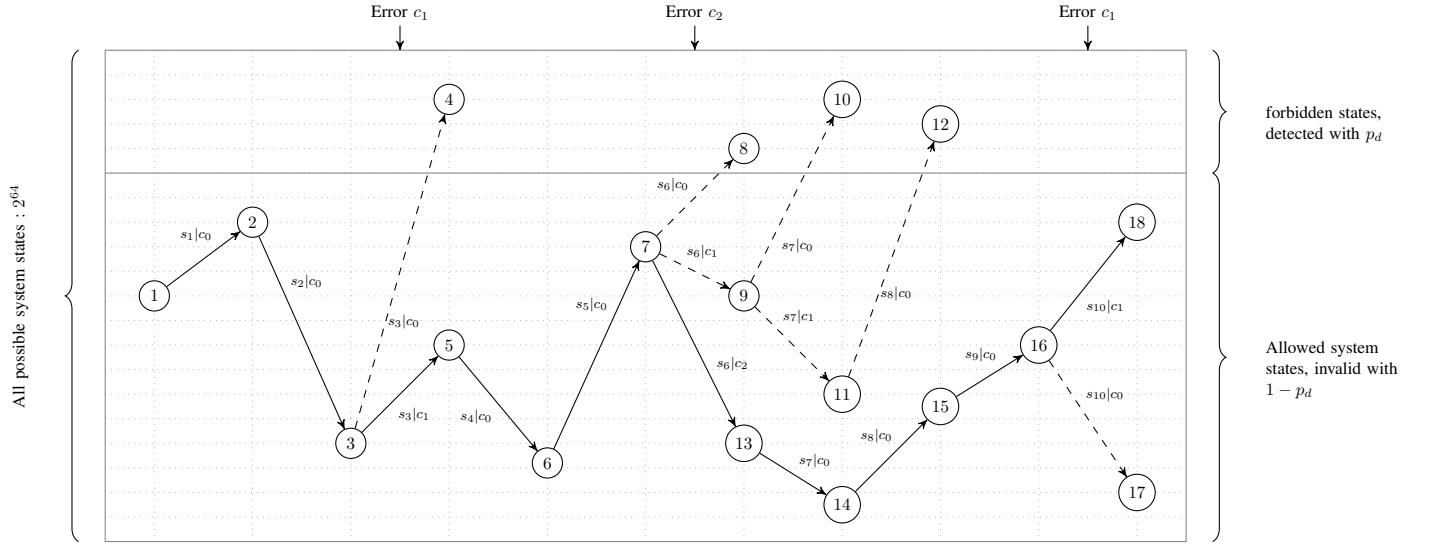


Fig. 1: Illustration of the forward error correction process and the corresponding system state transitions. Solid lines represent the proper correction path.

successive symbols (jn bits), it should contain asymptotically $2^{jnh(\epsilon)}$ leaves. If the probability of detecting an error is indeed p_d , on average only $(1 - p_d)^j$ of incorrect of them should survive the associated j redundancy tests. Then, the expected number of survivors can be asymptotically estimated as:

$$2^{jnh(\epsilon)}(1 - p_d)^j = \left(2^{nh(\epsilon) + \lg(\bar{p}_d)}\right)^j = \left(2^{nh(\epsilon) - R}\right)^j \quad (2)$$

Thus, codes with rates $R > R_0 := nh(\epsilon)$ or equivalently

$$p_d > p_d^0 := 1 - 2^{-R_0} \quad (3)$$

are correctable in the sense that the correction tree no longer grows exponentially. This is in fact the Shannon theoretical limit for the binary symmetric channel. Under this limit, a fixed large system state space is sufficient to guarantee that there are practically no state collisions during the correction process.

Selection of the most likely correction path is based on Bayesian analysis:

$$\Pr(E|O) \propto \Pr(O|E) \Pr(E)$$

where observation (O) in our case is the constructed tree. The explanation (E) we are looking for is the proper correction (leaf of the tree). We will denote such correction of j symbols ($J = nj$ bits) using the bit vector $(E_i)_{i=1}^J : E_i = 1$ if and only if the i^{th} bit is changed. The probability of the correction E can be directly calculated from the definition of the BSC as:

$$\Pr(E) = \epsilon^{\#\{i: E_i=1\}} (1 - \epsilon)^{\#\{i: E_i=0\}}$$

If the given correction (explanation) is proper, the tree nodes which are not on the correction path correspond to wrong

corrections. The probability of obtaining the current situation when E is indeed the proper path is

$$\Pr(O|E) = p_d^f (1 - p_d)^a$$

where f is the number of forbidden nodes outside the currently considered path and a is the number of allowed among them. By dividing this expression by an analogous one for all nodes of the tree, there remains only the dependency on the number of nodes in the currently considered correction path, i.e., $(1 - p_d)^{-j}$. Finally, by taking the logarithm of $\Pr(O|E) \cdot \Pr(E)$, we see that for a given correction tree, the next most probable correction path to consider corresponds to a node which maximizes:

$$\#\{i : E_i = 1\} \lg(\epsilon) + \#\{i : E_i = 0\} \lg(1 - \epsilon) - j \lg(\bar{p}_d) \quad (4)$$

The first two terms favor corrections with smaller number of corrected bits along the path. The last term favors longer paths. (4) represents the weight of a correction tree node and can be seen as a form of the Fano metric [7]. It is expressed as a logarithm of probability, therefore a difference of the weight of 1 denotes a node twice as probable than the other.

Although it is not the focus of this study, it is worth mentioning that this method can also be adapted to other types of communication channels. An erasure channel can be modeled by using $\epsilon = 1/2$ for the erased bits. The bit error probability can also be calculated in a per-bit manner to take into account soft-decisions of digital transmission demodulators. By increasing the set of possible symbol corrections, such an approach would also be capable of handling synchronization errors, such as bit deletion or duplication, which are difficult to deal with in other error correction methods.

In practice, (4) is calculated in a cumulative manner, i.e., the weight is calculated for the currently considered symbol only and it is added to the weight of its parent.

Algorithm 3 Forward state transition procedure

Require: s_{i-1} // Previous state
Require: y_i // New symbol
Require: $\mathbf{t} = t_1, t_2, \dots, t_{2^k}$ // State transition table
 $r_m \leftarrow 2^R - 1$ // Redundancy extraction mask
 $z \leftarrow t_{y_i >> R}$ // Temporary variable
 $c \leftarrow ((s_{i-1} \oplus y_i \oplus z) \& r_m) = 0$ // Redundancy verification result
 $s_i \leftarrow ((s_{i-1} \oplus z) >> R) + (s_{i-1} << (64 - R))$ // Apply state transition

Algorithm 4 Backward state transition procedure

Require: s_{i-1} // Previous state
Require: y_i // New symbol
Require: $\mathbf{t} = t_1, t_2, \dots, t_{2^k}$ // State transition table
 $r_m \leftarrow 2^R - 1$ // Redundancy extraction mask
 $z \leftarrow t_{y_i >> R}$ // Temporary variable
 $s_i \leftarrow ((s_{i-1} \oplus z) << R) + (s_{i-1} >> (64 - R))$ // Apply state transition

 $c \leftarrow ((s_i \oplus y_i \oplus z) \& r_m) = 0$ // Redundancy verification result

D. System State Transitions

This section describes the designed symbol-dependent system state transition procedure. The main design objective was to allow for low cost computation and to ensure good distance and statistical properties for correction in both directions.

A single transmitted symbol contains n bits of information, including k payload bits and R redundancy bits. In this study, we consider only natural R ; however, this derivation can be made more general, for example the redundancy can be added while entropy coding by introducing forbidden symbols of probability p_d to the original alphabet [4].

Upon reception of a symbol, the state of the system should be changed according to the received payload bits. For this purpose we perform a bit-wise exclusive disjunction of the current system state with a *transition vector* for the given payload. These transition vectors are constant throughout the process, and they are stored in a lookup table for the sake of operation performance. In the final step, the current state is shifted cyclically by R bits.

The system state update algorithm for forward correction is shown in Algorithm 3 and for backward correction in Algorithm 4.

Strictly pseudo-random initialization of the transition vectors is possible; however, a conscious choice enables significant improvements in the number of the decoding steps required. Firstly, since R least significant bits of the system state are directly used as the redundancy, it is of crucial importance that they allow for immediate detection of a forbidden system state. For this purpose, we evaluated all possible symbol correction patterns and selected R redundancy bits, so that the probability of missing these errors would be minimal. For example, for a code with $R = 1$ (code rate 7/8,) it is intuitively clear that the single bit of redundancy should be an exclusive disjunction on all of the payload bits. The corresponding matrix representation is:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

It immediately detects single bit errors, although it fails to detect double errors. For $R = 4$ (1/2 rate) the best matrix is:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

It detects all damages up to the triple bit and 56 of 70 quadruples. For $R = 6$ (1/4 rate) the matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}^T$$

immediately detects all up to the quadruple and 54 of 56 quintuple bit damages.

In order to maximize the amount of information carried by the redundancy bits, the number of payload bit sequences that correspond to 0 should be equal to the ones corresponding to 1 on each position of the remaining $64 - R$ bits of the transition vectors. Moreover, the remaining bits should be made as independent as possible. For this purpose, we divide the remaining $64 - R$ bits into k length segments, and initialize each segment with independent pseudo-random permutations of the possible 2^k payload bit sequences. This approach is a significant improvement on the pseudo-random choice of the whole transition vector. However, we believe that it could be optimized even further.

Together with the selection of the initial system state, utilization of pseudo-random numbers in the state transition process allows us to require the knowledge of the cryptographic key to utilize the introduced redundancy.

a) *Optimization of the Considered State Space:* In order to optimize the operation of the proposed decoder and severely reduce the amount of the considered correction patterns, we automatically discard the patterns which lead to incorrect system states. For each considered symbol, there exists only a small subset (approx. 2^{-R}) of correction patterns which lead to colliding redundancy bits of the posterior system state. For the sake of optimal performance, the decoder should consider only these corrections. The decoder pre-initializes a lookup table of such correction patterns and orders them by their weights. Such an approach successfully reduces the number of operations on the heap and significantly improves the correction performance.

In the forward decoding step the condition for the redundancy verification result c can be rewritten as:

$$(y_i \oplus s_{i-1}) \& r_m = t_{y_i >> R} \& r_m \quad (5)$$

Let us introduce a *modified symbol* z_i :

$$z_i = y_i \oplus (s_{i-1} \& r_m) \quad (6)$$

Applying the error mask to the modified symbol corresponds to applying it to the original symbol. However, it eliminates the dependency on the state from the original condition (5):

$$z_i \& r_m = t_{z_i >> R} \& r_m \quad (7)$$

This representation is more convenient to use in the lookup process. For example, we use it to generate the lookup table

of the allowed error patterns for each possible $z_i \in [0, 2^n - 1]$. Cyclic shift was modified to make it possible to use the same redundancy bits for both forward and backward correction. During the latter:

$$z_i = y_i \oplus (s_{i-1} \gg 64 - R) \quad (8)$$

E. Bidirectional Correction

Simultaneous forward and backward correction enables significantly better correction performance, as at least two CECs are required to cripple the correction process. When using bidirectional correction, the decoder builds two correction trees which are expanded independently as long as the symbol positions do not overlap. When this happens, the decoder looks for the identical system states for overlapping symbols. If this is successful, the correction paths are merged and the decoder yields the correct data stream.

In our implementation, we use a single array for storing the structure of both trees, and the decoder performs cyclically one correction step per direction. For each of the overlapping symbols, we create a binary search tree in order to obtain a logarithmic operation time. This way, the decoder can simply proceed with optimal expansion of both correction directions. An alternative approach would be to establish a barrier at the first overlapping symbol and make the decoder focus on finding the matching state there. However, the meeting point is usually placed asymmetrically in a CEC region and the resources are wasted in the direction which has already reached the proper correction. Then, the meeting provides almost no help with the difficult correction from the second direction. It is possible to shift the meeting point, although the available information is insufficient to do it optimally.

F. Final State

The proposed correction method requires to know the final system state in order to achieve the best correction performance. Without the final state, the decoder is not capable of backward correction. There is still the possibility of forward correction, which has a slightly lower efficiency when used alone.

The final state should be transmitted to the decoder in a reliable way. In this study, we do not address this issue. However, the decoder could easily be extended to support a small number of corrupted bits in the final system state. Instead of starting the backward correction from a single state, the heap of possible corrections should be pre-initialized with all reasonably probable correction patterns. Such a mechanism operates in the same way as ordinary correction of the transmitted symbols. For efficient operation, the weights of these alternative final states should be adjusted and calculated with some estimated bit error probability $\epsilon' \ll \epsilon$. The probability that a given bit should not be changed is $(1 - \epsilon')/\epsilon'$ times larger than the opposite hypothesis; therefore, for each changed bit, the initial weight of such modified states should be lower by $\lg((1 - \epsilon')/\epsilon')$.

For the sake of reliable transmission of this state information, an arbitrary error correction mechanism can be used.

However, since state-of-the-art correction algorithms are not efficient when working on small data blocks, we propose to aggregate the final states in large blocks and use the proposed correction mechanism. Alternative solution is placing the final states in the beginning of the succeeding frame (before encoding) - the vicinity to the initial state makes this region much more damage resistant. Such sequence of frames has to be decoded in backward order.

This additional communication overhead causes a slight drop in the effective code rate of the proposed method. For example if we assume that for rate 1/2 the state is protected using the same rate (on average $8 \cdot 2 = 16$ bytes), for 1024 byte frames the real rate is $512/(1024 + 16) \approx 0.4923$.

III. THEORETICAL ANALYSIS

In this section, we analyze the asymptotic statistical behavior of the proposed correction method. We quantitatively evaluate error concentrations using weight drops, as well as their probability distributions and average correction costs for BSC.

Both the decrease of the probability of error concentrations and the growth of the correction cost are usually exponential. Our analysis yields a theoretical rate of the exponents, allowing us to find the probability distribution of correction failure for a chosen step limit for both uni- and bidirectional correction. Our analysis is verified by experimental evaluation, described in detail in Section IV.

For better intuition with respect to the described behavior, it might be useful to experiment with an interactive simulator of the proposed correction tree approach [5].

A. Single-direction Sequential Correction

Asymptotically, the average weight (4) per node for a proper correction path is:

$$n\epsilon \lg(\epsilon) + n\tilde{\epsilon} \lg(\tilde{\epsilon}) - \lg(\tilde{p}_d) = -nh(\epsilon) - \lg(\tilde{p}_d)$$

Therefore, the condition of using a code rate below the Shannon limit ($1 - h(\epsilon) > \frac{k}{n} = \frac{n + \lg(\tilde{p}_d)}{n}$) is equivalent to that the weight of the proper correction path is statistically growing. Locally, however, it can decrease. In such situations, before finding the proper correction path again, the decoder needs to expand some of the wrong correction sub-tree.

The situation is outlined in Fig. 2. For the proper correction path, let us define the *weight drop* w for a given node as its weight minus the minimum from the weights of all future nodes, i.e. weight drop equals 0 if the weight does not become smaller. We will not use the node with this minimum weight until we expand trees of wrong corrections up to this weight drop. This means that it quantitatively describes error concentrations: not only the number of damaged bits, but also how densely they are distributed. The expected number of such wrong corrections that need to be considered will grow exponentially with w , but the probability of large w will drop exponentially with it.

Firstly, let us consider the weight drop probability distribution. Specifically, for an infinite correct path, let us define a

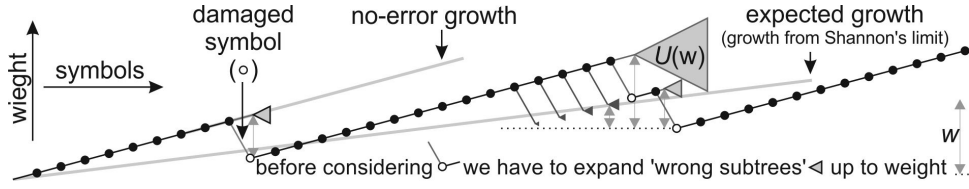


Fig. 2: Schematic diagram of the proper correction we aim to find. After an error, we expand the wrong corrections until reaching the weight of the proper one. We search for expected size and probability of occurrence for these wrong correction subtrees.

function $V(w)$ as the probability that the weight of further nodes (including this node) will drop by at most w . For $w < 0$, $V(w) = 0$. $V(0)$ corresponds to a situation in which the weight increases and it should be positive. This function is not continuous, but surprisingly it asymptotically tends to a continuous function.

Before analyzing a general case, let us focus for a moment on a simplified one. Let us assume that all symbols have exactly 1 bit ($n = 1$) of information. Observe that we can write V using its values for the previous position, getting a functional equation (9) for BSC.

If there was no such boundary behavior in $w = 0$, it would be a simple linear functional equation with a linear combination of exponents as a solution. Fortunately the functional equation is sufficient to find the asymptotic behavior.

We know that $\lim_{w \rightarrow \infty} V(w) = 1$, so let us assume that asymptotically

$$1 - V(w) \propto 2^{vw} \quad (11)$$

for some $v < 0$. Substituting it to (9), we get:

$$2^{vw} = \epsilon 2^{v(w+\lg(\epsilon)-\lg(\tilde{p}_d))} + \tilde{\epsilon} 2^{v(w+\lg(\tilde{\epsilon})-\lg(\tilde{p}_d))}$$

$$\tilde{p}_d^v = \epsilon^{v+1} + \tilde{\epsilon}^{v+1} \quad (12)$$

This equation always has a $v = 0$ solution, but for $p_d > p_d^0$ there emerges a second, negative solution which is of interest here. It can be easily found numerically and our simulations show that we are asymptotically reaching this behavior.

We can now go to the general case. By writing (9) analogously for a larger natural n , we get 2^n terms and after the substitution (11), we can collapse them:

$$2^{vw} = 2^{vw} \left(\epsilon 2^{v(\lg(\epsilon)-\frac{1}{n} \lg(\tilde{p}_d))} + \tilde{\epsilon} 2^{v(\lg(\tilde{\epsilon})-\frac{1}{n} \lg(\tilde{p}_d))} \right)^n$$

$$\tilde{p}_d^v = (\epsilon^{v+1} + \tilde{\epsilon}^{v+1})^n \quad (13)$$

Again we are interested in the $v < 0$ solution.

Next we need to estimate the asymptotic behavior of the size of the wrong correction sub-trees for large weight drops. Define $U(w)$ as the expected number of nodes of such a sub-tree, which would be constructed from a node of weight drop w . Each node of such a sub-tree can be thought of as a root of a new sub-tree. If we expand it for the corresponding weight drops, we obtain the expected number of nodes. Again,

$U(w) = 0$ for $w < 0$. For $w = 0$ we process this node, i.e., $U(0) \geq 1$.

First, let us focus on one bit blocks ($n = 1$) as previously. Connecting the node with its children we get the functional equation (10). We expect that for some $u > 0$ asymptotically

$$U(w) \propto 2^{uw} \quad (14)$$

By substituting (14) to (10) we obtain

$$2^{uw} = \tilde{p}_d \left(2^{u(w+\lg(\epsilon)-\lg(\tilde{p}_d))} + 2^{u(w+\lg(\tilde{\epsilon})-\lg(\tilde{p}_d))} \right)$$

$$\tilde{p}_d^{u-1} = \epsilon^u + \tilde{\epsilon}^u \quad (15)$$

This equation is very similar to (12). For $p_d > p_d^0$ we once again obtain two solutions, with the greater one equal to 1. Analogously to the previous analysis, due to strong boundary conditions in $w = 0$, we will be asymptotically reaching the smaller solution. This behavior is confirmed by our simulations. By comparing these two equations, we surprisingly obtain a simple correspondence between these two critical coefficients:

$$u = v + 1 \quad (v < 0, u > 0) \quad (16)$$

This relationship is also satisfied in the general case ($n > 1$). If ϵ does not match the channel's properties exactly, U would remain the same, but V would be slightly different. Therefore this relationship would be approximate only.

Having obtained the rate of the exponents $U(w)$ and $V(w)$ from (12) and (15), we can find the asymptotic behavior of the probability that a certain number of steps would be exceeded for a single node. Assume $U(w) \approx c_u 2^{uw}$, $V(w) \approx 1 - c_v 2^{vw}$ for some unknown coefficients c_u , c_v . Now the asymptotic probability that the number of nodes of a wrong correction sub-tree for a given proper node will exceed a certain number of steps s is:

$$\Pr \left(w > \frac{\lg(s/c_u)}{u} \right) \approx 1 - c_v 2^{\frac{v}{u} \lg(s/c_u)} = 1 - c_v \left(\frac{s}{c_u} \right)^{v/u}$$

$$\Pr(\# \text{ nodes in wrong subtree} > s) \approx 1 - c_p s^c \quad (17)$$

where $c := v/u = 1 - 1/u$ and $c_p = c_v/c_u^c$. Hence, we asymptotically obtain the Pareto probability distribution. The exponent, i.e. the shape of the distribution, can be found analytically. For code rate 1/2, the approximate values are:

$V(w) :=$ probability that the weight on the correct path will drop by at most w

$$V(w) = \begin{cases} \epsilon V(w + \lg(\epsilon) - \lg(\tilde{p}_d)) + \tilde{\epsilon} V(w + \lg(\tilde{\epsilon}) - \lg(\tilde{p}_d)) & \text{for } w \geq 0 \\ 0 & \text{for } w < 0 \end{cases} \quad (9)$$

$U(w) :=$ expected number of processed nodes of wrong correction from node of weight drop w

$$U(w) = \begin{cases} 1 + \tilde{p}_d (U(w + \lg(\epsilon) - \lg(\tilde{p}_d)) + U(w + \lg(\tilde{\epsilon}) - \lg(\tilde{p}_d))) & \text{for } w \geq 0 \\ 0 & \text{for } w < 0 \end{cases} \quad (10)$$

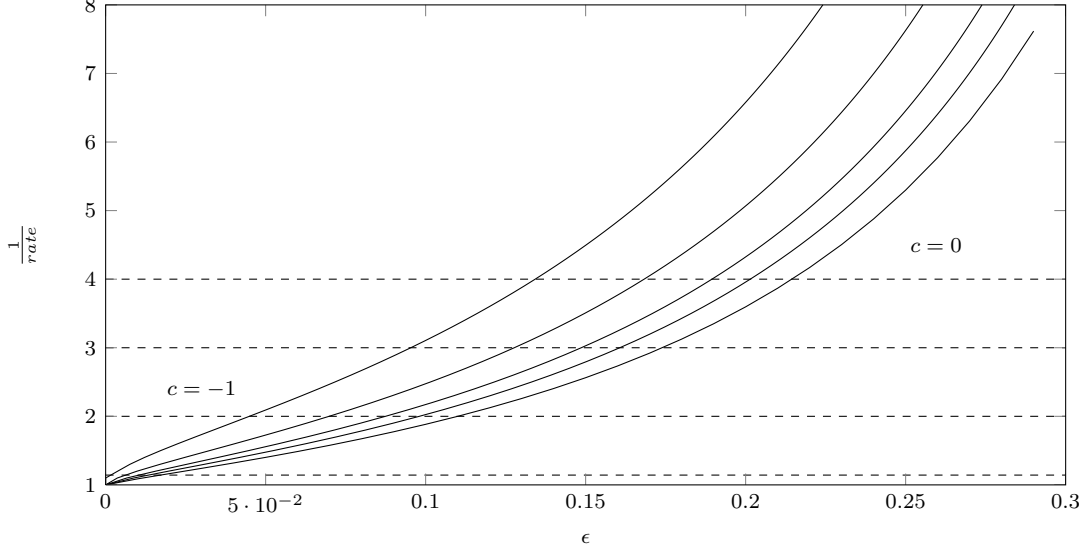


Fig. 3: Lines of constant c for different code rates - exhibiting similar behavior. From the left: the limit for the finite expected number of steps for unidirectional correction $c = -1$; for bidirectional correction $c = -\frac{1}{2}$; $c = -\frac{1}{4}$; $c = -\frac{1}{8}$; Shannon bound $c = 0$. The dashed lines represent the typical code rates.

| | | | | | | | | |
|------------|-------|-------|-------|------|-------|-------|------|------|
| ϵ | 0.03 | 0.045 | 0.06 | 0.07 | 0.08 | 0.09 | 0.10 | 0.11 |
| c | -1.46 | -1 | -0.67 | -0.5 | -0.35 | -0.22 | -0.1 | 0 |

This coefficient approaches zero in the Shannon limit ($\epsilon = 0.11$). For larger channel noises the weight function decreases statistically, therefore the weight drops would go to infinity. For lower channel noises, or if the errors were spread almost uniformly, the weight drops would remain close to zero, resulting in a small number of decoding steps.

The problem is that there can appear statistically large error concentrations which require a very large number of decoding steps for successful correction. If $c < -1$, their probability drops quicker than their influence, and the expected number of steps per node becomes finite. For $c \in [-1, 0)$ it becomes infinite, but since we are working on finite data frames, we are still able to perform the correction with a high degree of probability.

The numbers of decoding steps per node cannot be treated independently. The surrounding nodes have similar weight drops, and large values would implicate a large number of steps in the whole area around the considered node. The number of decoding steps grows exponentially with the weight drop, so in practice most of the steps should be located around the nodes with the largest weight drops (error concentrations).

The amount of memory available to the decoder limits

the number of steps that can be used to deal with an error concentration. A weight drop with a corresponding error concentration that cannot be corrected within a certain limit of steps is referred to as a CEC. The probability of not achieving this limit, i.e., that s steps are insufficient to correct a l node frame, is approximately

$$(1 - c_p s^c)^l \approx 1 - l c_p s^c$$

Increasing the number of steps a certain number of times, reduces the failure probability to approx. c -th power of this number of times. This allows us to reduce the failure probability to zero by simply increasing the number of the decoding steps. However, this is no longer practical near the Shannon limit. For example for $c = -0.1$, increasing the number of steps 2^{10} times would enable us to reduce the probability of failure only twofold.

A very high number of decoding steps increases the probability of system state collisions to a level where it cannot be safely ignored. The probability that two considered corrections up to a given symbol will correspond to the same system state is approximately proportional to the number of pairs, i.e. to the square of the width of the tree. However, the existence of such a collision does not necessarily mean that one of the two

involved corrections will be used. The only real problem which can leave a few dozens of bits damaged is the collisions with the proper correction patterns. Their probability is proportional to the width of the tree. Such collisions can occur on each position, therefore this probability is approximately proportional to the total number of steps. For a 64 bit state and $5 \cdot 10^7$ steps it is practically negligible ($2.6 \cdot 10^{-12}$) and it is difficult to imagine that, for example, a 128-bit state could turn out to be insufficient for any conceivable practical step limit. Below the Shannon limit, the size of the system state would need to grow logarithmically with the size of the frame, while for larger noises it would require to grow linearly to compensate for the lacking code rate.

B. Bidirectional Correction

The unidirectional correction will stop on a single CEC. In this case, on average only half of the frame is fully corrected. With bidirectional correction, the CEC would be approached from both sides, which in most cases allows for successful correction. During our experimental evaluation, more than 80% of such CECs were successfully corrected. Even if a single CEC cannot be dealt with, the data stream was successfully corrected up to the CEC from both sides, which means that only a few dozen known bits around the CEC remain unrepaired efficiently. As a result, for bidirectional correction, two CECs are required to essentially cripple the correction process. The probability that there will be at most one CEC is approx.:

$$(1 - c_p s^c)^l + l c_p s^c (1 - c_p s^c)^{l-1} \approx 1 - l^2 (c_p)^2 s^{2c}$$

The failure probability drops to approximately the square of the original, while the Pareto shape parameter approximately doubles. As a result, a ten-fold increase of the step limit reduces the probability of failure asymptotically 10^{-2c} number of times. For the 1/2 rate, the approximate values are:

| ϵ | 0.03 | 0.045 | 0.06 | 0.07 | 0.08 | 0.09 | 0.10 | 0.11 |
|------------|------|-------|------|------|------|------|------|------|
| 10^{-2c} | 846 | 99 | 22 | 10 | 5 | 2.7 | 1.6 | 1 |

The probability of failure also drops reversely proportional to approximately the square of the length of the frame (l), therefore the best performance is expected for short data frames. However, since each frame requires the final system state to be transmitted, shortening the frame would quickly result in a severe reduction of the code rate. As a result, a compromise is required.

A convenient way of comparing the theoretical results with the simulations is to order them by the number of the required decoding steps. The position in this order divided by their total number approximates the probability of performing a successful correction within the given step limit. As a result, we obtain an approximate distribution function of the required number of steps. This also allows for extrapolation of the bit error rate for lower channel noises. Based on the theoretical analysis presented, in the log-log scale it should approach the direction determined by $2c$. From Fig. 4 we see that it

exponentially approaches it at an unknown rate. However, the asymptotic direction corresponds well with the theory.

For low channel error rates, we were not able to achieve valid BER estimates from the experiments, as all simulated frames were transmitted correctly in a feasible computation time. In these cases, we use the described fits. In order to estimate the number of uncorrected bits with the insufficient number of decoding steps, we use the above probabilities of failure to calculate the probability of a single CEC and thus determine the probability distribution for the number of CECs. If there are m separate CECs, we should repair up to the first one from each side, what is on average approximately

$$\int_0^1 \int_0^1 \dots \int_0^1 \min(x_1, \dots, x_m) dx_1 \dots dx_m = \frac{1}{m+1}$$

of the whole frame; in this case, approx. $\frac{m-1}{m+1}$ bits should remain unrepaired.

IV. EXPERIMENTAL EVALUATION

This section describes the results of experimental evaluation of the proposed error correction method. We have implemented the described correction tree algorithm in C++. The sources are available at [1] under the GPL license.

The implemented codec is currently optimized for $n = 8$, $R \in \{1, 2, 3, 4, 5, 6\}$ and $n = 9$, $R = 6$ cases. However, it can be easily expanded to any case $n \in [2, 16]$, $R \in [1, n-2]$ by adding corresponding optimized definitions of immediately produced redundancy bits. In this study, R remains constant for all the blocks. Dynamic changes of R are a straightforward modification of the proposed scheme and allow for designing codes with an arbitrary rate. These n -bit blocks can be grouped into frames of an arbitrary size, as shown in Section II-F. In the experiments described, we use frame sizes l of 1024 and 4096. We use $n = 8$ and $R = \{1, 4, 6\}$ to obtain code rates of 0.875, 0.5 and 0.25 respectively. For the rate of 1/3 we use $n = 9$ and $R = 6$.

This study deals with reliable transmission over the BSC. The ϵ for the performed experiments has been carefully chosen for each case to cover the range where the codec ceases to provide reliable transmission capability. In each experiment, the codec transmits 1000 frames and collects the necessary statistics. Each frame is composed of a pseudo-random payload and the corresponding redundancy information appended by the encoder. The seed for the utilized pseudo-random number generator is changed on a per-frame basis.

The results obtained for the proposed correction method are presented and discussed in Section IV-A. A comparison with existing error correction codes is presented in Section IV-B.

A. Correction Tree Efficiency

The most important criterion for the efficiency of the proposed correction method is the number of the necessary decoding steps. The limit of the decoding steps, which stems from the available computational resources directly affects the correction performance. If the transmission channel does not generate any errors, the decoder performs only l steps. More

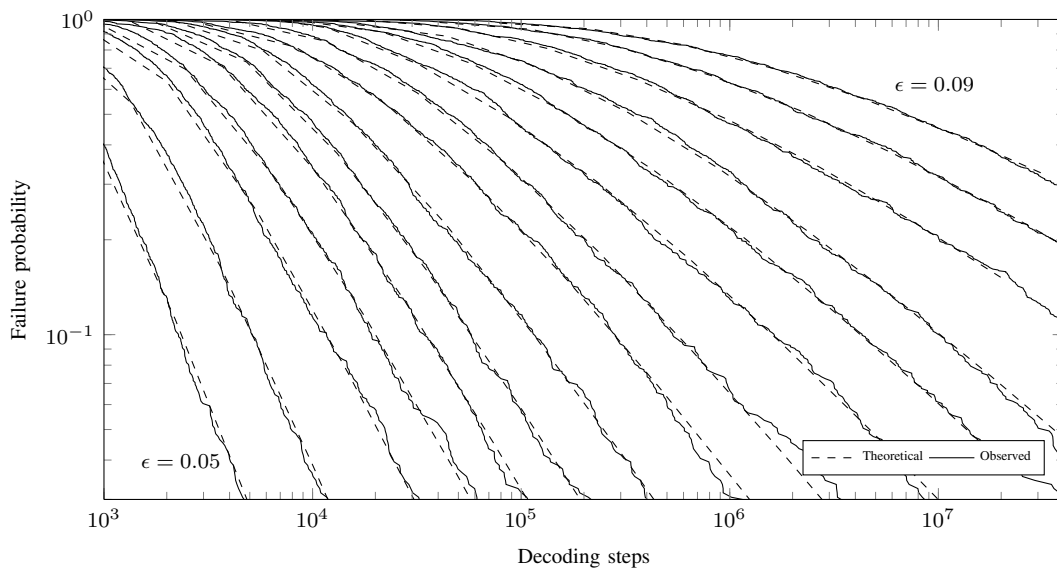
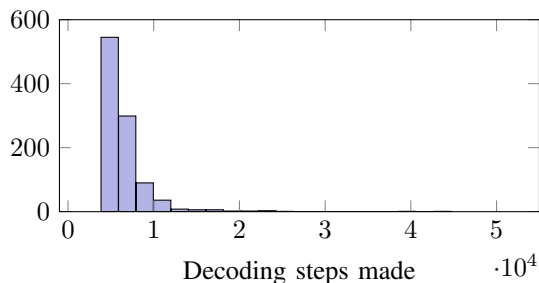
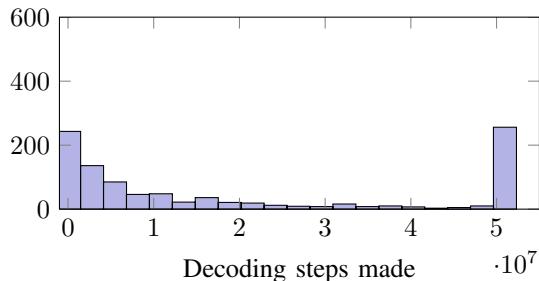


Fig. 4: Experimental results of the approximate distribution function of the number of decoding steps required for 1024 byte frames, the 1/2 rate (0.49) and channel BER successively 0.05, 0.055, 0.06, 0.0625, 0.065, 0.0675, 0.07, 0.0725, 0.075, 0.0775, 0.08, 0.0825, 0.085, 0.0875 and 0.09. In a log-log scale, it should approach the $2c$ direction - the dashed line is the result of fitting A, B, C coefficients of $2cx + A + Be^{-Cx}$ equation.



(a) $\epsilon = 0.045$



(b) $\epsilon = 0.080$

Fig. 5: The number of decoding steps performed by the decoder for the rate 1/2 code and $l = 4096$ blocks

decoding steps are needed when transmission errors occur. Figure 5 shows the histogram of the necessary decoding steps for $l = 4096$ and two example channels: $\epsilon = 0.045$ and $\epsilon = 0.08$. In case of the latter, the visible peak at the end of the histogram represents the frames damaged beyond repair. The remaining 75% of the frames have been successfully corrected with 10,367,638 decoding steps on average. The results from all of the performed experiments are collected in Table I.

The proposed algorithm allows for correction arbitrarily close to the Shannon limit. The only limitation is the amount of resources that are available on the receiving end of the transmission system. The maximum number of possible decoding steps stems directly from the available system memory and computational power. The size of the decoding node structure is 40B, and thus the memory requirements for step limits of $4 \cdot 10^5$, $2 \cdot 10^6$, $1 \cdot 10^7$ and $5 \cdot 10^7$ are 15MB, 76MB, 381MB and 1.9GB, respectively. At the cost of the decoding performance, it is possible to reduce the size of the correction tree node to approximately 20B. Figure 6 shows the achievable transmission error rates for these step limits for $l = 1024$ and $l = 4096$. On a standard PC with a single 3.16 GHz core, the correction process is performed at a speed of approximately 3.2 million steps per second. For smaller channel noise the processing speed reaches up to approx. 5 million steps per second. For larger ones it drops to an average 2 million steps per second. Combining these speeds with average steps per symbol from Table I, we see that this software implementation of the proposed approach is capable of processing up to a few megabytes per second on average for low noise levels. The processing speed drops gradually to a few hundred bytes per second, for extreme noise levels.

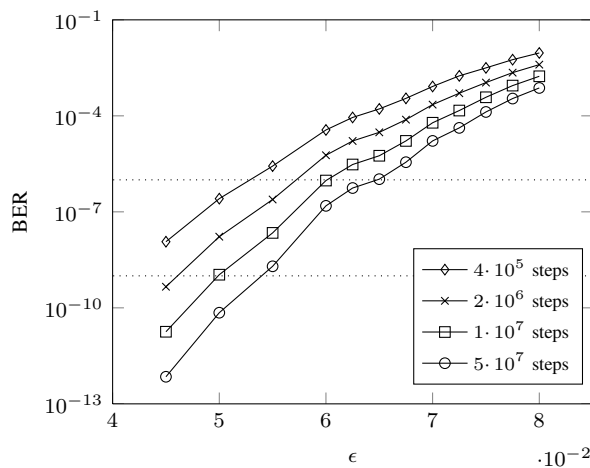
B. Comparative Evaluation

We compared the proposed correction tree algorithm with existing state-of-the-art forward error correction codes, namely the LDPC [8], [12] and turbo codes [3]. We implemented a BSC simulation scenario in C++ using the IT++ library [2]. The soft-information for the decoder input was calculated taking into account the log likelihood ratio (LLR) for the BSC channel, i.e. $\log_2 \frac{\epsilon}{1-\epsilon}$.

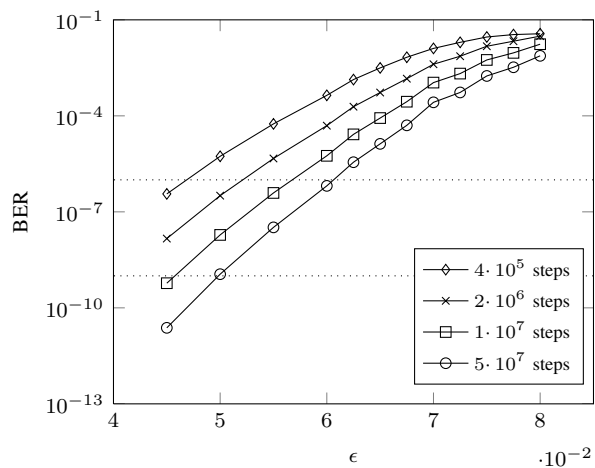
In this experiment, we consider two most popular coding

TABLE I: The achieved frame errors (FE) and the average number of decoding steps \bar{D} per symbol for the performed experiments (1 for pure decoding). The total number of simulated frames per setting is 1000.

| Rate $\frac{7}{8}$ | 0.002 | 0.003 | 0.004 | 0.005 | 0.006 | 0.007 | 0.008 | 0.009 | ϵ |
|--------------------|------------|------------|------------|-------------|------------|-------------|-------------|--------------|---------------------|
| $l = 1024$ | 0 1.274 | 0 1.702 | 0 2.882 | 0 5.447 | 0 18.76 | 1 133.9 | 0 232.6 | 14 1349 | FE \bar{D}/l |
| $l = 4096$ | 0 1.307 | 0 1.926 | 0 4.279 | 0 17.08 | 1 61.65 | 12 401.9 | 38 1199 | 150 3205 | FE \bar{D}/l |
| Rate $\frac{1}{2}$ | 0.05 | 0.06 | 0.065 | 0.07 | 0.075 | 0.08 | 0.085 | 0.09 | ϵ |
| $l = 1024$ | 0 2.385 | 0 8.946 | 0 21.80 | 0 76.92 | 5 680.7 | 25 2660 | 92 8232 | 273 18795 | FE \bar{D}/l |
| $l = 4096$ | 0 3.134 | 0 17.36 | 1 111.5 | 14 510.1 | 57 1776 | 229 4678 | 659 9645 | 945 11949 | FE \bar{D}/l |
| Rate $\frac{1}{4}$ | 0.13 | 0.14 | 0.15 | 0.16 | 0.165 | 0.17 | 0.175 | 0.18 | ϵ |
| $l = 1024$ | 0 1.873 | 0 2.988 | 0 7.972 | 0 21.12 | 0 89.90 | 0 193.7 | 5 705.0 | 16 2070 | FE \bar{D}/l |
| $l = 4096$ | 0 2.204 | 0 4.119 | 0 12.25 | 1 71.91 | 4 221.6 | 9 636.1 | 45 1658 | 220 4480 | FE \bar{D}/l |



(a) $l = 1024$



(b) $l = 4096$

Fig. 6: Performance of the correction tree algorithm for a $1/2$ rate code and different step limits. The dotted lines represent the commonly used thresholds for reliable transmission, i.e. BER of 10^{-6} and 10^{-9} .

rates: $\frac{1}{2}$ and $\frac{1}{3}$. The utilized LDPC codes were either generated randomly or taken from [11]. We used the belief propagation decoder. A summary of the LDPC codes used is shown in Table II.

For the turbo code, we used the (13,15) code from the Wideband Code Division Multiple Access (WCDMA) standard and the maximum a-posteriori probability (MAP) decoder with 10 iterations [2].

The results are shown in Fig. 7. The proposed correction algorithm delivers a very good correction performance. It outperforms the commonly available LDPC and turbo codes for small frame sizes and for higher channel error rates.

V. FURTHER PERSPECTIVES

The discussed correction process is usually very fast for practical settings and does not require large amounts of memory; however, with probability decreasing to zero, the

TABLE II: The utilized LDPC codes

| Code | l | K | Rate | Origin |
|----------|-------|-------|-------|------------------------------|
| MK 1920 | 1920 | 1280 | $1/3$ | 1920.1280.3.303 [11] |
| MK 4000 | 4000 | 2000 | $1/2$ | 4000.2000.3.243 [11] |
| MK 20000 | 20000 | 10000 | $1/2$ | 20000.10000.3.631 [11] |
| R 500 | 500 | 250 | $1/2$ | Randomly generated using [2] |

amount of required resources increases to infinity (Fig. 4). It suggests that high capacity decoders should be designed in a hierarchical way, where the stream is processed by many low memory decoders first. The frames would be passed to more powerful decoders only if they have failed.

In this section, we discuss the construction of extremely large memory decoders which are not limited by the amount of random access memory (RAM) on a single computer. We also discuss a generalization of the correction process for different

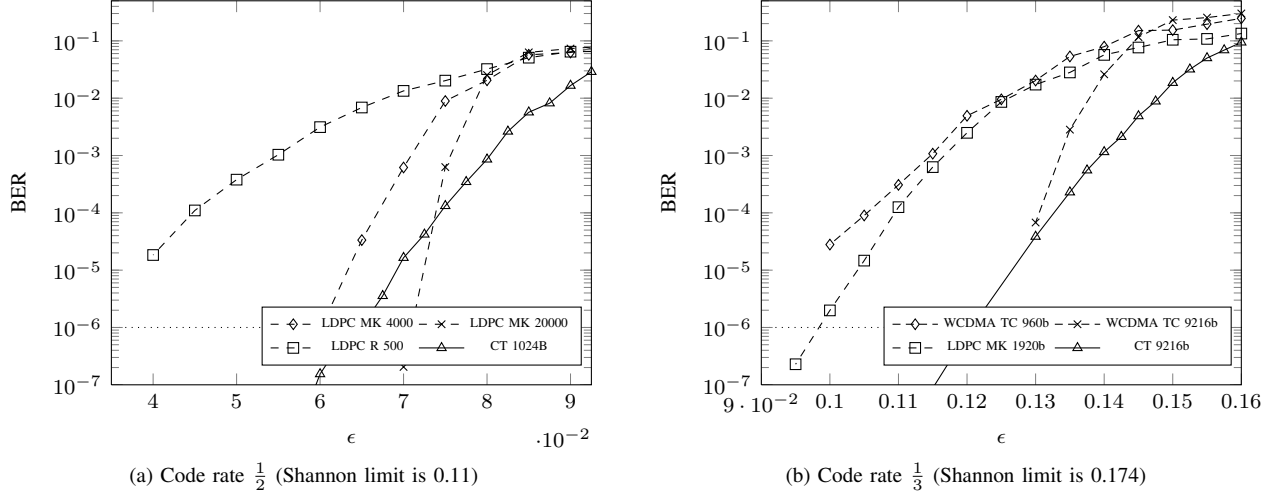


Fig. 7: Correction performance comparison with popular LDPC and turbo codes on a binary symmetric channel.

communication channels. Specifically, we consider the erasure channel, the additive white Gaussian noise (AWGN) channel, and channels with synchronization errors.

A. Handling Extremely Large Correction Trees

The presented algorithm was optimized for single core processing and within the capacity of RAM. In extreme situations, it may be necessary to allow for more decoding steps to increase the correction performance. For example using, a thousand times more steps for the $\frac{1}{2}$ rate and $\epsilon = 0.09$ would reduce the probability of failure about 20 fold. This section discusses the potential of generalizing the correction procedure to allow for efficient use of memory swapping or processing on multiple computers.

The original algorithm uses 3 large data structures: for storing the correction tree, the heap, and the binary search trees for each symbol position. Remembering the tree structure is useful for rapid application of the obtained correction path at a later stage, although it requires a lot of memory. While we would like to exchange data efficiently with the swapped memory segments or other computers, referring to such tables of nodes becomes problematic. However, the correction path can be retraced even without this structure. Having the proper state for any given position and a list of used states for the previous position, we can try successive corrections to make a step in the reverse direction until we get to a state from this list and so on.

If we no longer need the tree structure, the nodes can be handled independently, e.g. on different computers. The required information to be taken from the heap includes the symbol position, the system state before applying the correction, the currently considered correction of a given symbol, and the weight after applying this correction (in total, approx. 16B per tree node). In a single step, such a node would be retrieved from the heap and returned there with the updated correction pattern and the corresponding weight. Finally, the part of such a heap below a certain weight limit can be swapped for prospective consideration at a processing stage later.

When using multiple computers, each can have its own heap. Every node goes to only one such heap. Without communication, it is expected that the difference between the maximum weights of these heaps would increase. In order to prevent it, every few steps the considered nodes should be sent to the heap which has the lowest maximum weight in the cluster, instead of being stored in the local heap.

Additionally, we need to optimize the storage of the lists of the considered states for each symbol position. Originally used for stitching the two correction directions only, it would now also be used for retracing the proper correction path. We replace the previously used binary search trees with B-trees. They can be seen as a generalization, in which the nodes of the structure not only contain the state information and two pointers to its children, but become larger data blocks corresponding to a certain range of system states. Such a node contains a sorted list of system states from this range and potentially a sorted list of pointers to nodes corresponding to the sub-ranges. When a node exceeds its capacity, it is split into two parts and the structure of the B-tree needs to be updated. Using these large nodes, it is possible to insert a state or check if it is in the list by accessing only a few such data blocks. Therefore, such operations remain relatively fast, even when they are spread across the swapped memory or over the processing cluster.

In summary, working with extremely large trees involves restricting the operation of independent correction processors to the heaps which can be developed independently. Practical implementation of such a scheme would require a lot of work, but it may be useful for offline correction of critical data.

B. Other Channels and Applications

In this study, we have focused on the BSC. This section briefly discusses a generalization of the presented correction approach to different types of communication channels.

In the erasure channel, the received bits are either certain or completely lost. As such, this channel can be handled by sequential decoding, with the choice of $\epsilon = 0$ for undamaged

bits and $\epsilon = 1/2$ for the erased ones. The fact that all possible corrections up to a given symbol position are equally probable allows for a simpler correction approach, i.e. to store all corrections which pass the verification up to a given position, expand them to the succeeding position and so on. Denoting bit erasure probability by p_e , the analogous theoretical Pareto coefficient (19) for uni-directional correction and for $1/2$ rate are (derivation in Appendix):

| | | | | | | | | |
|-------|-------|-------|------|-------|-------|-------|-------|-----|
| p_e | 0.4 | 0.42 | 0.44 | 0.46 | 0.47 | 0.48 | 0.49 | 0.5 |
| c_e | -1.17 | -0.93 | -0.7 | -0.46 | -0.34 | -0.23 | -0.11 | 0 |

Fountain codes are a different class of erasure codes. They can produce an arbitrarily large number of data blocks, such that their undamaged sufficient portion allows us to reconstruct the original message. Such i^{th} data block can be constructed by concatenation of i^{th} redundancy bits from all symbols. This approach allows us to see it as an erasure channel with damages distributed in completely uniform way - without error concentrations and therefore with much better performance. Using correction trees would additionally allow us to retrieve almost all of the information also from damaged data blocks, which need to be ignored in standard fountain codes.

The presented algorithm can also be easily adapted to communication channels with ϵ being variable on a per bit basis, e.g. the AWGN channel. It is straightforward to include soft decoding information while calculating the weights of the successive nodes (4). Its technical inconvenience is that it can change the order of corrections for a single symbol, which slightly complicates the algorithm. Theoretical analysis in this case requires replacing equations such as (13) and (15) with corresponding integral equations.

The proposed sequential decoding can also handle different types of local errors by simply adding new types of child nodes. For example synchronization errors, such as bit deletion or duplication, which are difficult to handle with other correction methods. The previously considered BSC set of symbol correction patterns is a set of masks for appropriate bit flips. Consideration of bit deletion, for example, involves expanding the set of correction patterns to include all possible bit deletions with the weight decreased by a logarithm of deletion probability, correspondingly shifting succeeding symbols. It is also possible to detect and undo global bijective transformations from some assumed family. When simultaneously considering the same transformation with different parameters, the weights of the proper settings should quickly dominate the others. The initial weights should be chosen as minus logarithm of transformation probability.

VI. CONCLUSIONS

In this study, we presented a new approach to error correction using correction trees for efficient selection of possible correction patterns. The main contributions of our work include:

- using a much larger system state space and a new efficient coding paradigm optimized for this purpose,
- utilization of a bidirectional correction mechanism, which greatly reduces the probability of failure,

- adoption of a heap for correction pattern selection, which allows for logarithmic time in large tree access operations.

This study also describes numerous optimizations which allows for efficient practical implementation of the proposed correction approach. Our correction algorithm allows for practically complete correction with negligible processing time, as long as the channel error rate is lower than approx. half (for code rate $1/2$) the Shannon theoretical limit. For larger error rates, complete correction is still possible, but its cost in terms of computational complexity and memory requirements starts to increase rapidly. Commonly available error correction codes, such as LDPC or TC, are relatively costly for low error rates, and become unreliable as the channel noise increases, especially for small block sizes.

Due to the varying latency, the proposed method may not be well suited for real-time applications like audio or video calls. However, there are certain classes of applications which could benefit from the proposed correction approach:

- data storage - usually working on low error rate levels for which such correction is practically cost-free; however, time and random events can degrade the data, which could still be corrected provided that the degradation level is below the Shannon limit,
- far-space or underwater communication, which often transmits low amounts of data and where high correction cost can be easily afforded,
- authentication and reconstruction of digital content,
- applications where the redundancy should be used by authorized recipients only,
- communication channels with possible synchronization errors, like bit deletion or duplication.

APPENDIX

We will now find Pareto coefficients for the case of the erasure channel: we can be sure of bits in some positions, although the remaining ones are lost completely. Let us denote this probability of erasure by p_e . For a $J = jn$ bit long message, on average Jp_e bits are damaged; this means that 2^{Jp_e} possibilities remain, from which $(1-p_d)^J$ of the incorrect ones will survive the redundancy bit checks:

$$2^{jnp_e}(1-p_d)^j = \left(2^{np_e+\lg(\bar{p}_d)}\right)^j = \left(2^{np_e+k-n}\right)^j = \left(2^{k-n(1-p_e)}\right)^j \quad (18)$$

therefore we correct trees faster than they grow for $\frac{k}{n} < 1-p_e$, which is Shannon capacity again.

By choosing $\epsilon = 1/2$ for the lost bits, we could use sequential correction as previously. However, since there is no distinction between the allowed corrections of a given length, this time we can use a simpler and less demanding algorithm: remember all allowed corrections up to the given point, use such a list to find one for succeeding position and so on. The memory limit restricts the maximum number of nodes for a single position; with a nonzero probability, any such limit could be insufficient. Next we will find an analogous

$T(s) :=$ probability that the number of corrections per symbol is at most 2^s

$$T(s) = \begin{cases} p_e T(s - \lg(\tilde{p}_d) - 1) + (1 - p_e) T(s - \lg(\tilde{p}_d)) & \text{for } s \geq 0 \\ 0 & \text{for } s < 0 \end{cases} \quad (20)$$

coefficient as for BSC, relating the step limit increase with a decrease of probability of failure.

Denote by $T(s)$ the probability that logarithm of the number of corrections to consider in a single moment is at most s . We know that $T(s) = 0$ for $s < 0$, but $T(0) > 0$. Due to checking the redundancy bits, in each step the expected number of corrections is multiplied by $(1 - p_d)$. For each bit lost, this number is additionally doubled. Finally, for $n = 1$, we get functional equation (20). Assuming as previously that asymptotically

$$1 - T(s) \propto 2^{c_e s}$$

for some $c_e \leq 0$, we get

$$2^{c_e s} = p_e 2^{c_e(s - \lg(\tilde{p}_d) - 1)} + (1 - p_e) 2^{c_e(s - \lg(\tilde{p}_d))}$$

for general n this equation becomes:

$$\tilde{p}_d^{c_e} = (p_e 2^{-c_e} + 1 - p_e)^n \quad (19)$$

This c_e works as the c coefficient for BSC. It approaches 0 in the Shannon limit and effectively doubles for bidirectional correction.

ACKNOWLEDGMENT

We would like to thank Professor Andrzej Pach, from the Department of Telecommunications, AGH University of Science and Technology, for his guidance and help.

REFERENCES

- [1] C++ source code for the correction trees algorithm. <https://indect-project.eu/correction-trees/>.
- [2] The it++ library. <http://itpp.sourceforge.net/current/>. Version 4.2.
- [3] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Proc. of IEEE International Conference on Communications*, 1993.
- [4] J. Duda. Asymmetric numeral systems. arxiv: 0902.0271.
- [5] J. Duda. Correction trees, interactive wolfram mathematica demonstration. <http://demonstrations.wolfram.com/CorrectionTrees/>.
- [6] P. Elias. "coding for noisy channels. *ZRE Conv. Rept. Pt.* 4:37–47, 1955.
- [7] R. M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Transaction on Information Theory*, 9:64–73, 1963.
- [8] R. G. Gallager. *Low-Density Parity-Check Codes*. PhD thesis, Cambridge, MA, 1963.
- [9] I. M. Jacobs and E. R. Berlekamp. A lower bound on to the distribution of computation for sequential decoding. *IEEE Transactions on Information Theory*, IT-13:167–174, 1967.
- [10] D. J. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.
- [11] David J.C. MacKay. Encyclopedia of sparse graph codes. <http://www.inference.phy.cam.ac.uk/mackay/codes/data.html>.
- [12] David J.C. MacKay. Good error correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, pages 399–431, March 1999.